Pythonium

Release 0.1

Bruno Geninatti

Nov 15, 2021

CONTENTS:

CHAPTER

ONE

ABOUT PYTHONIUM

Pythonium is a space turn-based strategy game where each player leads an alienrace that aims to conquer the galaxy.

You must explore planets to search and extract a valuable mineral: the *pythonium*. This precious material allows you to build cargo and combat spaceships, or mines to get more pythonium.

Manage the economy on your planets, and collect taxes on your people to finance your constructions, but be careful! Keep your clans happy if you want to avoid unrest in your planets.

Put your space helmet on, set your virtualenv, and start coding.

Battle for pythonium is waiting for you!

CHAPTER TWO

INSTALLATION

You can install Pythonium by cloning the repository and running

```
$ python setup.py install
```

and then test your installation by running

```
$ pythonium --version
Running 'pythonium' version x.y.z
```

CHAPTER

THREE

SINGLE-PLAYER MODE

Once you have Pythonium installed you can test it for a single-player mode with some of the available bots. For example, the `standard_player` bot.

```
pythonium --players pythonium.bots.standard_player
```

Once the command finishes you should have a `<galaxy_name>.gif` file and a `<galaxy_name>.log`, where `<galaxy_name>` is a unique code generated for the game.

- `<galaxy_name>.gif`: This is an animation showing how the galaxy ownership changed along with the game, which planets belong to each player, ships movements, combats, and the score on each turn.
- `<galaxy_name>.log`: Contain the logs with all the relevant events during the game.

Here's an example of the gif

CHAPTER

FOUR

MULTIPLAYER MODE

Pythonium allows up to two players per game, and you can test it by providing two bots to the `--players` argument.

```
pythonium --players pythonium.bots.standard_player pythonium.bots.pacific_player
```

The output will be similar to the single player mode: one `.log` and one `.gif` file.

CHAPTER

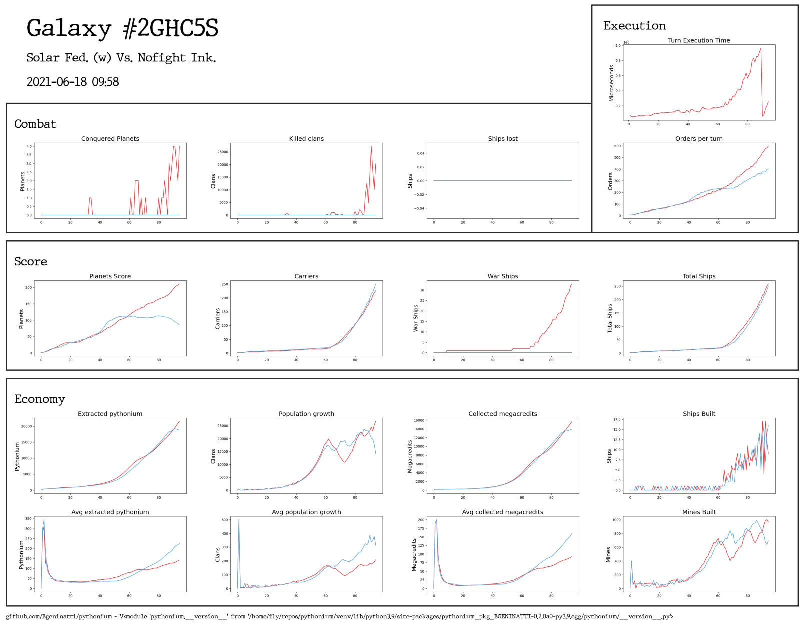
FIVE

METRICS

Providing the `--metrics` arguments, pythonium creates a report with several metrics of the game. This is especially useful to evaluate the performance of your players, and know their strengths and weaknesses.

```
pythonium --metrics --players pythonium.bots.standard_player pythonium.bots.pacific_
↪player
```

In addition to the `.gif` and `.log` now you will se a `report_<galaxy_name>.png` with several charts.



CHAPTER

SIX

ACKNOWLEDGE

This game is strongly inspired by VGA Planets, a space strategy war game from 1992 created by Tim Wisseman.

The modern version of VGA Planets is Planets.nu, and that project has also influenced the development of Pythonium.

To all of them, thank you.

CHAPTER SEVEN

WHAT NEXT?

Now you probably wants to write your own bot, didn't you?

Check out the *Tutorial* to see how to do it.

CHAPTER

EIGHT

TUTORIAL

8.1 Chapter 1 - The beginning of the road

Welcome player!

Welcome to the hard path of stop being part of a selfish colony of humanoids, jailed in their lonely planet with the only purpose of destroying themselves; to start being an adventurer, a space explorer, and a strategist. All with the power of your terminal and text editor.

In this section, you will learn how to create a player to play Pythonium.

Yes, you read correctly. You will not play Pythonium. You will build a player that to play Pythonium for you, and all your strategy must be implemented on that player.

This is a turn-based game, which means each player receives a set of information (or state of the game) at the beginning of turn *t*, and makes decisions based on that information to influence the state of the game at turn *t+1*. This sequence is repeated again and again in an iterative process until the game finishes.

Your player then is not more than a python class implementing a method that is executed every turn. This method receives as parameters the state of the `galaxy`, and some other `context` about the state of the game (i.e, the scoreboard and other useful data), and it must return the same `galaxy` instance with some changes reflecting the player's decisions.

Let's stop divagating and start coding.

Put this code inside a python file:

```python
from pyhtonium import AbstractPlayer

class Player(AbstractPlayer):

    name = 'Han Solo'

    def next_turn(self, galaxy, context):
        return galaxy
```

There are a few things to note from here.

In the first place, the `Player` class inherits from an `AbstractPlayer`. Second, there is one attribute and one method that needs to be defined in this class.

- `name`: The name of your player. Try to make it short or your reports and gif will look buggy.
- `next_turn`: A method that will be executed every turn. This is where your strategy is implemented.

8.1.1 Executing your player

Let's save now this file as `my_player.py` (or whatever name you like) and execute the following command:

```
$ pyhtonium --player my_player
** Pythonium **
Running battle in Galaxy #PBA5V2
Playing game..................................
Nobody won
```

The output will show the name of the galaxy where the game occurs, and some other self-explained information.

Once the command finishes, you will find in your working directory two files:

- `PBA5V2.gif`: A visual representation of the game. The closest thing to a UI that you will find in Pythonium.
- `PBA5V2.log`: A plain-text file containing all the information related to the game execution. Every change on the galaxy state (influenced by the player or not) is logged on this file.

Note: Notice that the name of both files is the galaxy name. Each game is generated with a random (kinda unique) galaxy name.

As a gif you will see something similar to this:

Reading from the top to the bottom:

- You are in the galaxy *#PBA5V2*
- You are Han Solo (your player's name)
- The turn at each frame is displayed at the left of the player name
- You have one planet and two ships
- Your planet and ships are in the blue dot. The rest of the dots are the others 299 planets in the galaxy.

Note: The blue dot is bigger than the white ones. The reason for this is that planets with any ship on their orbits are represented with bigger dots. This means your two ships are placed on your only planet.

Do you see it? Nothing happens. You just stay on your planet and do nothing for all eternity. By reviewing the player's code closely you will notice that this is precisely what it does: returns the galaxy without changing anything.

Congratulations! You just reproduced your miserable human life on earth, as a Pythonium player.

Wanna see the cool stuff? Then keep moving, human.

8.2 Chapter 2 - Understanding the unknown

Hello human. Good to see you again.

I understand that having you reading this is an expression of the desire of knowing more about the universe around you, and eventually leave your planet.

The next step is to know more about the galaxy. How many planets are? How far are they from you? Do you have starships? How can you use them?

Keep reading, and all your questions will be answered.

8.2.1 The galaxy, an introduction

In Pythonium, the `galaxy` is the source of all truth for you. It represents all your owned knowledge about the universe, and in most cases, all the information to develop your strategy will be extracted from the `galaxy`.

First, you need to learn what do you know about the galaxy. To do so we will use `ipdb`, the ancient oracle of python code.

This tool allows you to see what's going on with your python code at some point. In our case, we want to know what's going on at the beginning of each turn.

Note: Don't you know *ipdb*? Check it out.

Open the player you built in *Chapter 1* and set a trace in your `next_turn` method:

```
from pyhtonium import AbstractPlayer

class Player(AbstractPlayer):

    name = 'Han Solo'

    def next_turn(self, galaxy, context):
        import ipdb; ipdb.set_trace()
        return galaxy
```

Once executed you will see something similar to:

```
          8     def next_turn(self, galaxy, context):
          9         import ipdb; ipdb.set_trace()
---> 10          return galaxy

ipdb> _
```

Note: If you don't remember how to do execute your player check on *Executing your player*

Now we can start investigating the `galaxy`.

```
ipdb> galaxy
Galaxy(size=(500, 500), planets=300)
```

Ok then, this means you are in a galaxy of 500 light-years width and 500 ly height (`size=(500, 500)`) compounded by 300 planets (`planets=300`).

There are three main galaxy attributes that you must know in deep.

`turn`

Your time reference. The turn that is being played.

```
ipdb> galaxy.turn
0
```

As expected, the game just began, and you are in turn 0.

To move one turn forward, use the `c` command.

```
ipdb> c

         8     def next_turn(self, galaxy, context):
         9         import ipdb; ipdb.set_trace()
---> 10         return galaxy

ipdb> galaxy.turn
1
```

Note: And as you may suspect, there is no way to come back in time. Time always moves forward.

`planets`

This attribute stores the state of all the planets in the galaxy.

`galaxy.planets` is a python dictionary where the keys are planet's *`Position`*, and the values are *`Planet`* instances.

```
ipdb> type(galaxy.planets)
<class 'dict'>

ipdb> pp galaxy.planets
{(2, 124): Planet(id=ecf5f0b9-d639-48fb-ac06-cb0027d03d5b, position=(2, 124),␣
↪player=None),
 (3, 466): Planet(id=b20406cb-b764-4842-8dac-ec13c2038ca9, position=(3, 466),␣
↪player=None),
 (4, 129): Planet(id=ec53e2a9-24e2-49f5-aa56-4a6337b06b87, position=(4, 129),␣
↪player=None),
 (4, 294): Planet(id=40712b86-5bf3-453f-9714-760dbe771570, position=(4, 294),␣
↪player=None),
...
}

ipdb> len(galaxy.planets)
300
```

Note: In the previous example, we use the `ipdb` command `pp`, as an alias for pprint.

A planet has tons of attributes, for now we will focus just in a few of them:

- `id` a unique identifier for the planet,
- `position` is the planet position in the galaxy in (x, y) coordinates,
- `player` is the planet's owner, it can be `None` if the planet is not colonized or the owner is unknown to you.

ships

In a similar way as with the planets, the `galaxy.ships` attribute is a python list that stores references to every *`Ship`* in the galaxy.

```
ipdb> type(galaxy.ships)
<class 'list'>

ipdb> pp galaxy.ships
[Ship(id=b615699e-c70e-4e55-b678-fb0513abbb0b, position=(27, 23), player=Han Solo),
 Ship(id=5b8e15a8-a319-43d0-bdd0-6be675d1742e, position=(27, 23), player=Han Solo)]

ipdb> len(galaxy.ships)
2
```

The ships, also have `id`, `position`, and `player` attributes.

From `galaxy.ships` output we can tell there are two known ships in the galaxy, and both are yours (notice the `player=Han Solo`).

8.2.2 Querying to the galaxy

The `galaxy` has methods that allow you to filter `ships` and `planets` based on several criteria. In this section, we will present some receipts to answer common questions that you may have.

Where are my planets?

By looking carefully into the `galaxy.planets` output you will find a planet with `player=Han Solo`.

That's your planet!

But you may be thinking there should be an easier way to find which planets are yours (if any). And there is: this can be done with the *`Galaxy.get_player_planets`* method.

This method takes a player name as attribute and returns an iterable with all the planets where the owner is the player with the name you asked for.

```
ipdb> my_planets = galaxy.get_player_planets(self.name)
ipdb> pp list(my_planets)
[Planet(id=1fa89759-6834-478a-9eda-6985dd95a0c7, position=(27, 23), player=Han Solo)]
```

Note: You can access to the name of your *`Player`* inside your `next_turn` method with the `self.name` attribute.

Where are my ships?

In a similar fashion to planets, you can find all your ships with the *Galaxy.get_player_ships* method.

```
ipdb> my_ships = galaxy.get_player_ships(self.name)
ipdb> pp list(my_ships)
[Ship(id=b615699e-c70e-4e55-b678-fb0513abbb0b, position=(27, 23), player=Han Solo),
 Ship(id=5b8e15a8-a319-43d0-bdd0-6be675d1742e, position=(27, 23), player=Han Solo)]
```

In single-player mode *Galaxy.get_player_ships* always returns all the ships in `galaxy.ships`, as there are no abandoned ships in pythonium (with `player=None`).

But in multiplayer mode, you can also find enemy ships in the `galaxy.ships` attribute. In that case, this function can be handy to get only your ships, or the visible enemy ships.

Are there ships on my planet orbit?

Let's suppose you want to transfer some resource from one planet to another, the first thing you want to know is if there is any ship in the same position as your planet, to use this ship to transfer the resource.

This can be answered with the *Galaxy.get_ships_in_position* method.

This method takes a position as parameter and returns an iterable with all the known ships in that position.

In our case, that will be the `position` attribute of your planet.

```
ipdb> my_planets = galaxy.get_player_planets(self.name)
ipdb> some_planet = next(my_planets)
ipdb> some_planet
Planet(id=1fa89759-6834-478a-9eda-6985dd95a0c7, position=(27, 23), player=Han Solo)

ipdb> ships_in_planet = galaxy.get_ships_in_position(some_planet.position)
ipdb> pp list(ships_in_planet)
[Ship(id=b615699e-c70e-4e55-b678-fb0513abbb0b, position=(27, 23), player=Han Solo),
 Ship(id=5b8e15a8-a319-43d0-bdd0-6be675d1742e, position=(27, 23), player=Han Solo)]
```

Is my ship in a planet?

Now think the opposite example, you have a ship and you want to know if it is located on a planet or in deep space.

This can be answered by simply searching if there is planets in the ship's position.

```
ipdb> my_ships = galaxy.get_player_ships(self.name)
ipdb> some_ship = next(my_ships)
ipdb> some_ship
Ship(id=b615699e-c70e-4e55-b678-fb0513abbb0b, position=(27, 23), player=Han Solo)

ipdb> planet = galaxy.planets.get(some_ship.position)
ipdb> planet
Planet(id=1fa89759-6834-478a-9eda-6985dd95a0c7, position=(27, 23), player=Han Solo)
```

8.2.3 Turn `context`

Apart from `galaxy` there is a second argument received by the `Player.next_turn` method: the turn `context`.

The `context` contains additional metadata about the turn and the overall game.

```
ipdb> type(context)
<class 'dict'>

ipdb> context.keys()
dict_keys(['ship_types', 'tolerable_taxes', 'happypoints_tolerance', 'score'])
```

Here we see that `context` is a dictionary with several keys. For now, we will focus on the `score`.

```
ipdb> context['score']
[{'turn': 1, 'player': 'Han Solo', 'planets': 1, 'ships_carrier': 2, 'ships_war': 0,
↪'total_ships': 2}]
```

From the score we know:

- The current turn number is `1`,
- there is only one player called 'Han Solo' (that's you!),
- Han Solo owns,
  - one planet,
  - two carrier ships
  - zero warships,
  - and two ships in total

This is, in fact, consistent with the found results in previous sections. When you query your owned planets, the result was one single planet, and for your ships, the result was two ships.

You can verify that both ships are carriers by doing

```
ipdb> for ship in my_ships:
    print(ship.type.name)

carrier
carrier
```

In the next chapters, we will explore a bit more about the `context`, different ship types, and their attributes.

How to exit from the *ipdb* debugger

Pythonium has a special command for exit the `ipdb`. You will notice that the usual `exit` command will not work in this case. Exiting from the infinite loop of time is a bit more complex.

If you want to exit the debugger do:

```
ipdb> from pythonium.debugger import terminate
ipdb> terminate()
```

8.2.4 Final thoughts

In this chapter, we explained how to access the different objects from the galaxy, with a focus on those objects owned by your player. Depending on the complexity of the player that you want to implement, you might find useful one method or another. That is something you need to discover yourself, but it is good to have an overview.

You can also implement your own query methods for `galaxy.planets` and `galaxy.ships` depending on your needs. For starters space explorers, the methods presented in this section should be enough for most cases.

In the next chapter, you will learn how to move your ships.

Keep moving human, the battle for pythonium is waiting for you.

8.3 Chapter 3 - Han Solo: The Random Walker

Hi human. Glad to see you here. I thought you were lost in some capitalist leisure streaming service.

In this chapter, you will learn how to move once for all from your primitive planet and explore the galaxy. Once completed this tutorial you will be a globetrotter on the galaxy. The Han Solo of the Pythonium universe.

Warning: If you don't know who Han Solo is stop here and come back once you were watched the full original trilogy of Star Wars.

8.3.1 `target`: Where do you want to go?

Each `ship` has a `target` attribute indicating where the ship is going. This is one of the control variables for your ships. You can edit this parameter to order your ships to go to some specific point in the galaxy.

Start the `ipdb` debugger as you learned in *Chapter 2*, and select some random ship to be your explorer:

```
ipdb> my_ships = galaxy.get_player_ships(self.name) # Find all your ships
ipdb> explorer_ship = next(my_ships) # Select the first ship
```

Now let's see where the explorer ship is going:

```
ipdb> print(explorer_ship.target)
None
```

This means the ship has no target. In the next turn, it will be in the same position.

You can verify it easily.

```
ipdb> galaxy.turn # Check the current turn
0
ipdb> explorer_ship.position # Check the ship's position
(43, 37)
ipdb> c # Move one turn forward
...
ipdb> galaxy.turn # Now you are in turn 1
1
ipdb> my_ships = galaxy.get_player_ships(self.name) # Find the explorer ship again
ipdb> explorer_ship = next(my_ships)
ipdb> explorer_ship.position # The ship position is the same as previous turn
(43, 37)
```

The ship stays in the same position when time moves forward. It is not going anywhere.

Note: Note that when you move one turn forward `ipbb` do not save the variables declared in the previous turn. That's why we need to search the `explorer_ship` again.

8.3.2 Knowing the neighborhood

The next step is to find a destination for the `explorer_ship`

For sure you want to visit one of the many unknown planets around you (those with `player=None`), and possibly you don't want to travel for all eternity. We need to find some unknown planet near yours to arrive fast. The ship should arrive by the next turn.

But wait a minute, how fast the `explorer_ship` moves?

Every ship has a `speed` attribute indicating how many light-years can travel in a single turn.

```
ipdb> explorer_ship.speed
80
```

Based on this we can say the ship can travel up to 80ly in a single turn. The next step is to find an unknown planet that is 80ly or less from your planet.

The *`Galaxy.nearby_planets`* method allows you to find all the planets that are up to a certain distance away (or less) from a specific position. This method takes a `position` and a distance (called `neighborhood`) and returns a list with all the nearby planets around that position.

In our case, the neighborhood will be 80ly, the distance the ship can travel in one turn, and the position will be the ship location.

```
ipdb> neighborhood = galaxy.nearby_planets(explorer_ship.position, explorer_ship.
↪speed)
ipdb> pp neighborhood
[Planet(id=7d9321ab-57cb-4a05-afaa-c2f4ef8e4627, position=(43, 37), player=Han Solo),
 Planet(id=a374a560-ba94-43b1-87b0-78eca8ca5b97, position=(25, 41), player=None),
 Planet(id=e3319ed0-24ec-491c-bb76-a418d9b8b508, position=(112, 50), player=None),
 Planet(id=1b7d714e-22d2-4ca2-826a-bf0656138793, position=(115, 9), player=None),
 Planet(id=70279963-541b-49c9-bb87-32cf6936f45f, position=(31, 42), player=None),
 Planet(id=73f25d86-44f1-4cfc-a8ac-44a96affa1d9, position=(9, 21), player=None),
 Planet(id=1c7ec1c3-7aea-44bf-b582-1f7e3cb3b7ec, position=(81, 27), player=None),
 Planet(id=1378a7ab-2120-46d3-ac93-fc50632141b0, position=(96, 62), player=None),
 Planet(id=fb0d019d-ca71-4353-a06c-d3b4898ffd82, position=(93, 44), player=None),
 Planet(id=02539d23-2911-4354-81f5-9a1f83ef0936, position=(21, 86), player=None),
 Planet(id=38ce324b-ce2a-4bf1-997c-bb8990ae7509, position=(67, 37), player=None),
 Planet(id=4e19fda6-ac81-4d85-bdde-bd7244430a2e, position=(70, 33), player=None),
 Planet(id=e2234771-dbeb-425f-9b0a-1e761f5cf3e1, position=(44, 18), player=None),
 Planet(id=b5b025dd-dfcf-4ca5-8b03-67bb3a04479f, position=(30, 92), player=None),
 Planet(id=4b29c3d8-3c2f-4b33-8ca7-f451eb269e21, position=(61, 110), player=None),
 Planet(id=72b77b24-0063-42f1-aeb0-259f04125cbd, position=(67, 71), player=None),
 Planet(id=bf00cfa3-aece-48e6-8d67-11b3797e2f2c, position=(42, 69), player=None),
 Planet(id=43bcb3bb-b788-46e9-b425-8539caeff03c, position=(89, 64), player=None),
 Planet(id=0a9f5a40-034e-4fe8-a6b1-83f3437e09c8, position=(109, 54), player=None),
 Planet(id=a51d8923-1003-4357-bb2b-f3efa7d5023e, position=(17, 35), player=None),
 Planet(id=da112184-1e01-41ee-b146-d073946ce41e, position=(32, 81), player=None),
 Planet(id=765a19df-2639-4efd-8aa6-30ff3926039c, position=(75, 40), player=None),
 Planet(id=40052c15-3ffa-4dfa-ad22-9afbd0a16091, position=(95, 57), player=None)]
```

Cool, right?

All those planets are one turn away the `explorer_ship`. Notice that your planet is included in the neighborhood (because your ship is located in it and the distance to it is zero).

8.3.3 Traveling

Now let's select the target for the ship. For now, keep it simple: pic some random unknown planet from the list.

```
ipdb> unknown_nearby_planets = [p for p in neighborhood if p.player is None]
ipdb> import random
ipdb> target_planet = random.choice(unknown_nearby_planets)
ipdb> target_planet
Planet(id=1b7d714e-22d2-4ca2-826a-bf0656138793, position=(115, 9), player=None)
```

That's your ship first destination. An unknown planet one turn away from your ship's location.

The next step is set the ship's `target` as the planet's `position` and move one turn forward.

```
ipdb> galaxy.turn # Check the current turn
1
ipdb> explorer_ship.position # Check the ship position
(43, 37)
ipdb> explorer_ship.target = target_planet.position # set the ship target
ipdb> c # move one turn forward
```

Where is the ship now?

```
ipdb> galaxy.turn # you are one turn ahead
2
ipdb> my_ships = galaxy.get_player_ships(self.name) # Find all your ships
ipdb> explorer_ship = next(my_ships) # And keep the explorer ship
ipdb> explorer_ship.position # Check the ship position
(115, 9)
ipdb> explored_planet = galaxy.planets.get(explorer_ship.position) # Find the planet␣
↪in the ship's position
ipdb> explored_planet
Planet(id=1b7d714e-22d2-4ca2-826a-bf0656138793, position=(115, 9), player=None)
```

Your explorer ship just arrived at the target planet. A new and unknown rock in the middle of the space with a lot of things to learn about and explore.

Congratulations human. You did it. You left the pathetic rock where you spent your whole life, and now you are in a different one. Probably more pathetic, probably more boring, maybe you don't even have air to breathe or food to eat. But hey… you are a space traveler.

8.3.4 Putting the pieces together

In this chapter, we explained how to move your ships. You learned the first, and most basic command: Ship movement.

But we also developed a strategy. I call it "The Random Walker Strategy": A group of ships moving around, exploring planets without much more to do but travel around the galaxy.

Let's *exit the debugger*, edit your player class, and apply the random walker strategy to all your ships.

You will end up with something like this:

```python
import random
from pythonium import AbstractPlayer

class Player(AbstractPlayer):

    name = 'Han Solo'

    def next_turn(self, galaxy, context):
        # Get your ships
        my_ships = galaxy.get_player_ships(self.name)
        # For every of your ships...
        for ship in my_ships:
            # find the nearby planets...
            nearby_planets = galaxy.nearby_planets(ship.position, ship.speed)
            # pick any of them...
            target_planet = random.choice(nearby_planets)
            # an set the target to the selected planet
            ship.target = target_planet.position

        return galaxy
```

After executing your player the generated gif should look similar to this one:

Can you see those ships moving around? That, my friend, is what I call freedom.

8.3.5 Long travels

The implemented random walker strategy moves ships to planets that are one turn away from the original position only.

If you send a ship to a point that is furthest the distance the ship can travel in one turn (this is `ship.speed`), it will take more than one turn to arrive at the destination. In the next turn, the ship will be at some point in the middle between the target and the original destination.

Of course, you can change the ship's target at any time during travel.

Note: **Challenge** Build a random walker player that travels to planets that are two turns away only (and not planets that are one turn away)

8.3.6 Final thoughts

In this chapter we introduced the *`target`* attribute, and how it can be used to set a movement command for a ship.

We also explained how to find planets around certain position with the *`Galaxy.nearby_planets`* method.

Finally, this chapter is a first attempt to describe a player-building methodology in pythonium. Usually, you will make use of the debugger to test some commands, try a few movements and see how they work from one turn to another. This will help you to start a draft for your player strategy, and after that, you will need to code it in your player class.

The debugger is a good tool for testing and see how things evolve in a rudimentary way. On more complex players it is hard to track all the changes and commands that happen in one turn. Imagine you having an empire of more than 100 planets and around 150 ships, it is impossible to check all the positions and movements with the `ipdb` debugger.

For those cases, there are more advanced techniques of analysis that involve the generated logs and the report file. But that is a topic for future chapters.

I hope to see you again, there’s still a lot more to learn.

CHAPTER

NINE

PLAYER API REFERENCE

All the classes, methods, and attributes described in this sections can be used in your `next_turn` method.

Despite some of these classes have additional methods and attributes not listed here, those are not useful for you in any way.

You shouldn't expect any useful information from those methods and attributes. Most of them are used by Pythonium internally.

class pythonium.**AbstractPlayer**

name: str
Player's name. Please make it short (less than 12 characters), or you will break the gif and reports.

next_turn(*galaxy*, *context*)
Compute the player strategy based on the available information in the `galaxy` and `context`.

Parameters

- **galaxy** (`Galaxy`) – The state of the Galaxy known by the player.
- **context** (`dict`) – Aditional information about the game.

Each player sees a different part of the galaxy, and the `galaxy` known by every player is different.

A galaxy contains:

- All his ships and planets,
- All the enemy ships in any of his planets,
- All the enemy ships located in the same position as any of his ships,
- All the attributes of a planet that has no owner if a player's ship is on the planet,
- The position of all the planets (not the rest of its attributes),
- All the explosions that occur in the current turn.

The player won't know the attributes of enemy ships or planets but the position.

`context` has the following keys:

- `ship_types`: A dictionary with all the ship types that the player can build. See: *ShipType*
- `tolerable_taxes`: The level of taxes from where `happypoints` start to decay.
- `happypoints_tolerance`: The level of happypoints from where * `score`: A list with the score for each player.
- `turn`: Number of the current turn.

Return type `Galaxy`

class `pythonium.`**`Galaxy`**(*name*, *size*, *things*, *explosions=None*, *turn=0*)

Galaxy of planets that represents the map of the game, and all the known universe of things.

Parameters

- **size** (`NewType()(Position, Tuple[int, int])`) – Galaxy height and width
- **things** (`List[`*`StellarThing`*`]`) – Stellar things that compound the galaxy
- **explosions** (`Optional[List[Explosion]]`) – Known explosions in the galaxy
- **turn** (`int`) – Time in galaxy

static compute_distance(*a*, *b*)

Compute the distance in ly between two points (usually two `position` attributes).

Parameters

- **a** (`NewType()(Position, Tuple[int, int])`) – Point of origin to compute the distance
- **b** (`NewType()(Position, Tuple[int, int])`) – Point of destination to compute the distance

Return type `float`

distances_to_planets(*point*)

Compute the distance between the `point` and all the planets in the galaxy.

Return type `Dict[NewType()(Position, Tuple[int, int]), float]`

get_ocuped_planets()

Return all the planets colonized by any race

Return type `Iterable[Planet]`

get_planets_conflicts()

Return all the planets in conflict: Planets with at least one enemy ship on it

Return type `Iterable[Tuple[Planet, List[Ship]]]`

get_player_planets(*player*)

Returns an iterable for all the known planets that belongs to``player``

Return type `Iterable[Planet]`

get_player_ships(*player*)

Returns an iterable for known ships that belong to `player`

Return type `Iterable[Ship]`

get_ships_by_position()

Returns a dict with ships ordered by position. Ships in the same positions are grouped in a list.

Return type `Dict[NewType()(Position, Tuple[int, int]), List[Ship]]`

get_ships_conflicts()

Return all the ships in conflict: Ships with, at last, one enemy ship in the same position

Return type `Iterable[List[Ship]]`

get_ships_in_deep_space()

Returns an iterable for all the ships that are not located on a planet

Return type `Iterable[Ship]`

get_ships_in_planets()
Return a list of tuples `(planet, ships)` where `planet` is a *Planet* instance and `ships` is a list with all the ships located on the planet

Return type `Iterable[Tuple[Planet, List[Ship]]]`

get_ships_in_position(*position*)
Returns an iterable for all the known ships in the given position

Return type `Iterable[Ship]`

property known_races
List all the known races that own at least one ship or one planet.

Return type `Set[str]`

nearby_planets(*point*, *neighborhood*)
Return all the planets that are `neighborhood` light-years away or less.

Return type `List[Planet]`

search_planet(*_id*)
Return the planet with ID `id` if any

Return type `Planet`

search_ship(*_id*)
Return the ship with ID `id` if any and is known

Return type `Ship`

class pythonium.**Planet**(*position*, *id=NOTHING*, ***, *temperature*, *underground_pythonium*, *concentration*, *pythonium*, *mine_cost*, *player=None*, *megacredits=0*, *clans=0*, *mines=0*, *new_mines=0*, *new_ship=None*, *taxes=0*)
A planet that belongs to a *Galaxy*

Represent a planet that can be colonized.

The `temperature` on the planet will determine how happy the `clans` can be (temperature defines the `max_happypoints` for the planet).

Each planet has some amount of `pythonium` on the surface that can be used by the colonizer race, and some `underground_pythonium` (with a certain `concentration`) that needs to be extracted with `mines`.

The higher the `taxes` are over `cfg.tolerable_taxes`, the faster the `happypoints` decay.

The lower the `happypoints` are over `cfg.happypoints_tolerance`, the lower the planet production of `clans`, `pythonium`, and `megacredits` will be (in proportion of `rioting_index`).

can_build_mines()
Computes the number of mines that can be built on the planet based on available resources and `max_mines`

Return type `int`

can_build_ship(*ship_type*)
Indicates whether `ship_type` can be built on this planet with the available resources

Return type `bool`

clans: int
Amount of clans on the planet

concentration: float
Indicates how much pythonium extract one mine. It is always between zero and 1.

property dclans

Aditional clans for the next turn. The absolute change in the population considering:

- `rioting_index`: Negative influence,
- `cfg.max_population_rate`: Positive influence,
- `cfg.max_clans_in_planet`: Positive influence,
- `clans`: Positive influence

Return type `int`

property dhappypoints

Absolute change on happypoints for the next turn

Return type `int`

property dmegacredits

Absolute change in megacredits for the next turn considering:

- `taxes`: Positive influence
- `rioting_index`: Negative influence,
- `clans`: Positive influence,
- `taxes_collection_factor`: Positive influence

Do not consider `new_mines` and `new_ship` cost, or ship transfers.

Return type `int`

property dpythonium

Absolute change in pythonium for the next turn considering:

- `mines`: Positive influence,
- `concentration`: Positive influence,
- `rioting_index`: Positive influence.

Do not consider `new_mines` and `new_ship` cost, or ship transfers.

Return type `int`

happypoints: int

The level of happyness on the planet.

This has influence on `rioting_index`

max_happypoints: int

The maximum level of happypoints that the population of this planet can reach.

It is based on the planet's temperature.

Its maximum value (100) is reached when the planet's temperature is equal to `cfg.optimal_temperature`

property max_mines

The maximum number of mines that can be build in the planet

Return type `int`

megacredits: int

Amount of megacredits on the planet

mine_cost: pythonium.vectors.Transfer
Indicates the cost of building one mine.

mines: int
Amount of mines on the planet

new_mines: int
Attribute that can be modified by the player

New mines that the player order to build in the current turn. This value is set to zero when the player orders are executed.

new_ship: pythonium.ship_type.ShipType
Attribute that can be modified by the player

The new ship that the player order to build in the current turn.

This value is set to `None` when the player orders are executed

player: str
The owner of the planet or `None` if no one owns it.

pythonium: int
Pythonium in the surface of the planet. The available resource to build things.

property rioting_index
If the `happypoints` are less than `cfg.happypoints_tolerance` this property indicates how much this unhappiness will affect the planet's economy.

i.e: if `rioting_index` is 0.5 pythonium production, and megacredits recollection will be %50 less than its standard level.

Return type `float`

taxes: int
Attribute that can be modified by the player

Taxes set by the player. Must be between zero and 100.

The level of taxes defines how much megacredits will be collected in the current turn.

If the taxes are higher than `cfg.tolerable_taxes` the planet's happypoints will decay `planet.taxes - cfg.tolerable_taxes` per turn.

See *`dmegacredits`*

temperature: int
The temperature of the planet. It is always between zero and 100.

underground_pythonium: int
Amount of pythonium under the surface of the planet, that needs to be extracted with mines.

class pythonium.**Ship**(*position*, *id=NOTHING*, *player=None*, ***, *type*, *max_cargo*, *max_mc*, *attack*, *speed*)
A ship that belongs to a race.

It can be moved from one point to another, it can be used to move any resource, and in some cases can be used to attack planets or ships.

attack: int
Indicates how much attack the ship has. See *`ShipType.attack`*

clans: int
Amount of clans on the ship

max_cargo: int
Indicates how much pythonium and clans (together) can carry the ship. See *ShipType.max_cargo*

max_mc: int
Indicates how much megacredits can carry the ship. See *ShipType.max_mc*

megacredits: int
Amount of megacredits on the ship

pythonium: int
Amount of pythonium on the ship

target: NewType.<locals>.new_type
Attribute that can be modified by the player

Indicates where the ship is going, or `None` if it is stoped.

transfer: pythonium.vectors.Transfer
Attribute that can be modified by the player

Indicates what resources will be transferred by the ship in the current turn.

If no transfer is made this is an empty transfer.

See *Transfer* bool conversion.

type: pythonium.ship_type.ShipType
Name of the *ShipType*

class pythonium.**ShipType**(*name*, *cost*, *max_cargo*, *max_mc*, *attack*, *speed*)
Defines the attributes of a ship that the player can built.

attack: int
Attack of the ship. It will be used to resolve conflicts.

cost: pythonium.vectors.Transfer
Transfer instance that represents the cost of a ship of this type.

max_cargo: int
Max cargo of clans and pythonium (together) for this ship.

In other words, you should always expect this:

```
>>> ship.megacredits + ship.clans <= ship.max_cargo
True
```

Megacredits do not take up physical space in the ship so are not considered for `max_cargo` limit.

max_mc: int
Max amount of megacredits that can be loaded to the ship.

In other words, you should always expect this:

```
>>> ship.megacredits <= ship.max_mc
True
```

name: str
A descriptive name for the ship type. i.e: 'war', 'carrier'

class pythonium.**Transfer**(*megacredits=0*, *pythonium=0*, *clans=0*)
Represent the transfer of resources from a ship to a planet and vice-versa, or the cost of structures. This second case can be considered as a transfer between the player and the game.

Some useful properties of this class are:

Negation

Changes the direction of the transfer.

```
>>> t = Transfer(clans=100, megacredits=100, pythonium=100)
>>> print(-t)
Transfer(megacredits=-100, pythonium=-100, clans=-100)
```

Addition and Subtraction

Add/subtracts two transfers

```
>>> t1 = Transfer(clans=100, megacredits=100, pythonium=100)
>>> t2 = Transfer(clans=10, megacredits=10, pythonium=10)
>>> print(t1 + t2)
Transfer(megacredits=110, pythonium=110, clans=110)
```

Multiplication and Division

Multiplies/divides a transfer with an scalar

```
>>> t = Transfer(clans=100, megacredits=100, pythonium=100)
>>> print(t*1.5)
Transfer(megacredits=150, pythonium=150, clans=150)
>>> print(t/1.5)
Transfer(megacredits=150, pythonium=150, clans=150)
```

Empty transfers

The conversion of a transfer to a boolean return `False` if the transfer is empty

```
>>> t = Transfer()
>>> print(bool(t))
False
```

clans: int
Amount of clans to transfer

megacredits: int
Amount of megacredits to transfer

pythonium: int
Amount of pythonium to transfer

class pythonium.**Explosion**(*ship*, *ships_involved*, *total_attack*)
A ship that has exploded because of a conflict

Provides some insights about which ship exploded and where

Parameters

- **ship** (`Ship`) – Destroyed ship
- **ships_involved** (`int`) – Amount of ships involved in the combat
- **total_attack** (`int`) – Total attack involved in the combat

ship
Ship that has been destroyed.

ships_involved
Amount of ships involved in the combat.

total_attack
Total attack involved in the combat.

class pythonium.core.**Position**(*x*)

class pythonium.core.**StellarThing**(*position*, *id=NOTHING*, *player=None*)

id: <module 'uuid' from '/home/docs/.pyenv/versions/3.7.9/lib/python3.7/uuid.py'>
Unique identifier for the *StellarThing*

player: str
The owner of the `StellarThing` or `None` if no one owns it.

position: NewType.<locals>.new_type
Position of the *StellarThing* in (x, y) coordinates.

INDEX